Rapport de projet

Environnement de Programmation

BONNET Thomas , JAJOUX Jeremy

1 Fonctions de base du jeu

Soit les fonctions de grid.c

Ici la plupart des fonctions étaient simples à programmer car elles sont de constructions moindres et ont un objectif unique et court.

Les fonctions ayant le plus posé de problèmes furent les fonctions relatives aux mouvement des tuiles, c'est à dire "can_move" et "do_move", car ces dernières nécessitent de prendre en compte toutes les différentes situations possibles lors d'un mouvement ou d'une fusion de tuile.

C'est pourquoi, lors des premières ébauches du code, toutes ces fonctions étaient très longues et répétitives afin d'inclure toutes les possibilité sans se soucier de la structures de celles-ci.

L'étape suivante a donc consistée à réduire la quantité de code et à créer des fonctions annexes, comme "move" ou "tile_fusion", pour alléger le code de chaque fonction et permettre une élimination des répétitions avec des sous-fonction dotés de différents arguments pour fonctionner selon la situation plutôt qu'une section de code répétée avec de légères modifications.

Par exemple dand la sous-fonction suivante, les variables x et y qui parcourent la grille peuvent êtres interchangées afin de parcourir la grille de bas en haut et de droite à gauche. Les variables i et j repèrent la cellule de destination et les variables ii et jj la cellule vide suivante dans le cas où un mouvement est possible mais pas une fusion.

```
static int
move(grid g,int x,int y,int i,int j,int ii,int jj){
  int a=0;
  if(get_tile(g,x,y)!=0){
    if(get_tile(g,i,j)==0)
      move_tile(g,x,y,i,j);
    else{
      if(get_tile(g,x,y)==get_tile(g,i,j))
tile_fusion(g,x,y,i,j);
      else
move_tile(g,x,y,ii,jj);
      a=1;}
  }
  return a;
}
```

On arrive alors à un code relativement compact et un peu illisible si on ne l'as pas écrit malheureusement.

Un autre obstacle s'est manifesté dans la fonction "add_tile", cette dernière nécessite de prendre en compte toutes les cellules vides et ajouter de manière aléatoire dans une de ces cellules la valeur 2 ou, dans 10% des cas, la valeur 4.

Pour ce faire il a fallut créer un tableau de cellules vides en parcourant la grille, puis à l'aide de la fonction "random" séléctionner une des cases du tableau correpondant à une cellule vide au hasard. On ajoute ensuite une tuile de valeur décidée également à l'aide de random. L'utilisation de random s'est faite à tâton, plus particulièrement l'initialisation de la seed en tandem avec la fonction "time" qui devait se trouver avant l'éxecution du jeu dans son entièreté.

Au final ce fut une partie du projet qui a nécessité beaucoup de temps et de changement pour fonctionner comme désiré et de manière la plus optimisée possible.

2 Fonctions pour faire fonctionner le jeu

Les fonctions contenues dans play.c

Cette partie du code a été relativement compliquée à faire fonctionner et à nécessité un grand nombre de changement que ce soit pour améliorer son fonctionnement, optimiser son code ou ajouter de nouvelles options.

Plus que la création et l'affichage de la grille qui sont d'une écriture simple, le réel problème fut de faire fonctionner la grille comme voulut en entrant une touche. Plusieurs problèmes se sont accumulés pour ce problème précis, tout d'abord il fallait réussir à lire une touche entrée par l'utilisateur, ceci à été accomplis à l'aide de la fonction "getchar" qui, couplée à la fonction "getDirection" pour sélectionner la direction en fonction de la touche préssée, à permis de faire fonctionner le jeu correctement. Cependant "getchar" renvoie l'un après l'autre la touche préssée et la touche Entrée utilisée pour la validation, ce qui faisait faire deux fois le même mouvement à la grille. Nous avons pallié à cela en ajoutant à l'execution du programme une variable booléenne "change" qui permettait de vérifier qu'une touche de direction valide venait d'être préssée et ne l'execute qu'une fois. Ce problème n'est pas étonnament pas apparut dans la version ncurses du jeu, probablement dû à un fonctionnement différent de "getch".

Un autre problème, qui a été plus rapide à régler, était de faire en sorte que le programme termine bien le jeu. Pour cela il a fallut à nouveau ajouter une variable boléenne "playing" qui voit sa valeur changer en accord avec la fonction "game_over" après chaque execution de mouvement et stoppe la fonction lorsque plus aucun mouvement n'est possible.

```
void
play_terminal(grid g){
  bool playing=true;
  bool change=true;
  int c;
  dir d;

  afficher_grille_terminal(g);
  while(playing){
```

```
    c=getchar();

    if(validDirection(c)){
      d = getDirection(c);
      change=true;
    }

    if(c == TX)
      playing = false;

    else if(can_move(g,d) && change){
      play(g,d);
      afficher_grille_terminal(g);
      playing=!game_over(g);
    }
    change=false;
  }
}
```

Le programme est également doté d'un peu de code après la fin du jeu qui permet d'afficher la fin du jeu, le score obtenu et la plus grande valeur de tuile obtenue dans le but de faciliter de futures tests d'efficacité sur les stratégies.

3 Les Tests

Trouvés dans test.c

4 Le Makefile

Il y a peut de chose à dire ici car nous n'avons pas beaucoup d'expérience dans la création d'un Make fonctionnel, cependant il permet de compiler le programme du jeu, les test et la librairie sans problème.

Il est un peu regrettable que nous n'ayons pas fait en sorte que tout soit disposé dans différents dossier et tout les fichiers et la compilation doivent se faire à la racine.

5 Les Stratégies

Probablement la partie la plus infructueuse du projet, la stratégie que l'on a ici a été créée à partir d'un concept simple et contre toute attente marche correctement, les stratégies rapides et efficientes sont les mêmes avec seulement une différence de profondeur.

La stratégie par du fait que le score est changé lors d'une fusion et augmente de la valeur de la nouvelle tuile. Ainsi on peu supposer que si le score est grand, on possède également des tuiles de valeur importantes, puisqu'on veut atteindre la tuile 2048 c'est intéressant. Cette stratégie execute donc une simple récursion en jouant les quatres direction possibles sur autant de copie de la grille actuelle et conserve la direction qui aboutit au meilleur score. On ajoute ici une varible "depth" qui répète cette opération pour autant de mouvement à la suite qu'indiqué par sa valeur.

Ceci ne devrait pas très bien marcher dû à la nature aléatoire du placement de tuile, ainsi un mouvement prédit par la fonction obtiendra le même score que le mouvement réel mais tout mouvement subséquents provoqué par la recursion n'auras pas nécessairement la même valeur que le mouvement réel. En effet la version efficiente de la stratégie qui possède une valeur de depth plus élevée est moins efficace que la version rapide, ceci vient du fait qu'en augmentant le nombre de récursion on augmente le nombre de fausses prédictions.

Cependant faute de meilleure stratégie c'est celle-ci que nous conservons.

6 Interface Graphique

Le corps du fichier 2048.c